Undocumented Express

William G. Brown
Senior Consulting, Symmetry Corp.

This article first appeared in Oracle Internals, February 2000.

In 1995, Oracle Corporation realized that their product offerings lacked the functionality of a multidimensional database. Moving quickly to fill this critical need, they examined writing a new multidimensional database while evaluating the major multidimensional vendors' offerings. The purchase of the Express multidimensional database and applications from Information Resources, Inc. made them a market leader in the growing business intelligence market.

Express has greatly expanded Oracle's ability to deliver business intelligence to the user community. Coupled with Oracle's strong relational database and query tools, Express and its related applications (Oracle Sales Analyzer and Oracle Financial Analyzer) fill a critical gap in Oracle's solution for on-line analytical processing.

Express' strength lies in its flexibility of design and rich Stored Procedure Language (SPL). What follows is a description of a series of undocumented Express tips and functions I have found useful over the years precisely because they take advantage of Express' flexibility.

Undocumented Express Tips and Functions

The tips and functions can be classified into these nine categories:

- MCALC
- Worksheets
- Time dimensions
- TCONVERT
- Capstones
- XCA
- Faster SQL-based loaders
- Custom functions
- Conjoints

MCALC

MCALC is an undocumented function in Express written for Oracle Sales Analyzer (OSA). While officially MCALC is only supported with OSA, developers have used it for years in custom Express applications as well. MCALC is used to calculate custom aggregates; i.e., user defined dimension members that have not been pre-calculated.
The MCALC function is attached to a formula that the users will access for reporting. There are five

components to MCALC: variable, screenby, method, members, and weight.

The basic syntax for MCALC has the following format:

```
MCALC(variable, SCREENBY boolean method OF members, WEIGHTBY weight)
```

An example that uses the MCALC syntax follows.

```
define F.SALES formula decimal <GEOG PROD TIME>
Eq MCALC(v.sales, SCREENBY if V.SALES eq NA THEN NO else YES, TOTAL of
MEMBERS.G, TOTAL of MEMBERS.P, TOTAL of MEMBERS.T)
```

Now let's look at each MCALC component, its definition, and a specific example for each component.

Variable

The data variable (or formula) users are reporting on. For example:

```
define V.SALES variable decimal
<PRODGEOG <PROD GEOG> TIME>
```

Screenby

This is an optional variable or formula that controls the handling of NA's and calculations. Information Resources, Inc. added SCREENBY to OSA before OSA was acquired by Oracle. SCREENBY supports the type of census data Information Resources sells and is normally used to distinguish a product that is unavailable in a market from a product that is not sold in a market. Generally, in custom applications this option is set to return a NO if the variable is NA and YES if the variable is not NA. For example:

```
SCREENBY if V.SALES eq NA then NO else YES
```

This setting will allow custom aggregates to be calculated, but NA's will be reported as NA's and will not be converted to ZERO on reports. However, if any cell of a custom aggregate is NA, the custom aggregate will be NA.

Method

Six methods are available to calculate a custom aggregate. The possible values for METHOD are: AVERAGE, FIRST, LAST, LARGEST, TOTAL and NONADD.

- AVERAGE: A simple or weighted average (see option WEIGHTBY).
- FIRST: The first value in the member list. This is typically used with time dimensions.
- LAST: The last value in the member list. This is typically used with time dimensions.
- LARGEST: The largest value from the member's list. If all values are negative, MCALC returns a

zero.
- TOTAL: Sums the values from the member's list.
- NONADD: Returns NA in all cases.

Member List

This is a text variable dimensioned by one of the formula's base dimensions. For pre-calculated dimension members, this variable contains an NA value. For custom aggregates, this variable contains a list of dimension members that will be used to calculate the aggregate value. For example:

```
define MEMBERS.P variable text <PROD>
limit PROD to 'MYCUSTAGG'
rpr down PROD MEMBERS.P
PROD MEMBERS.P
MYCUSTAGG PROD_BIKE
PROD_CAR
PROD_WAGON
```

If the Express option MCALCINT is set to YES (default), Express expects integer values representing dimension member positions. I recommend setting MCALCINT to NO and using dimension members in the member list. In the above example, the custom aggregate MYCUSTAGG is made up of dimension members PROD_BIKE, PROD_CAR, and PROD_WAGON, instead of the integer values 1,2, and 3 that would have been the default values had MCALCINT been set to YES. This way, as new members are added, the member list does not have to be updated to reflect possible changes in the order of the dimension members.

Weightby

This is an optional variable or formula that allows for weighted averages. Omitting this argument assigns all dimension members a weight of one. If a weight is NA, it is omitted from the custom aggregate. If WEIGHTBY is used, the following formula is used to calculate the average:

```
Sum of(weight*value*sign of the number)/
sum of(weight * sign)
```

MCALC is a complex function. If you wish to use MCALC, please experiment with it until you fully understand its abilities and limitations. In the meantime, to help you get started, below I've listed several MCALC limitations and some methods to mitigate them.

Dynamic Member Lists

Once built, a member list is static and remains unchanged until someone manually updates it. To generate a dynamic member list, you can use a formula. This is useful in situations where there are two sources for the

member list, such as a system where some members on the member list were generated by the system and other members were generated by an end user.

We can fool MCALC using Express' versatile text manipulation tools. For example,

```
Define MEMBERS.P formula text <PROD>
Eq uniquelines(joinlines(USERMEM.P, SYSMEM.P))
```

Display of NA's and ZEROS

Discussed briefly in the SCREENBY section of this article, the display of NA's and zeros is often confusing with MCALC. Using the SCREENBY option, MCALC can either:

- Translate all NA's to ZERO, which translates all NA's on a report to zeros. NA and ZERO have considerably different meanings so this behavior is not desirable.
- Not translate NA to ZERO, however if any member in the custom aggregate is NA, the custom aggregate will result in an NA.
- Treat NA as ZERO only for custom aggregates, which provides a solution that combines the best of both options by allowing you to report NA's as NA's and correctly calculate custom aggregates using zeros when appropriate.

Using the example from the members list section above, we can illustrate each of the above behaviors:

Translate all NA's to ZERO

```
dfn F.SALES_ZERO formula dec <PROD GEOG TIME>
eq Eq MCALC(v.sales, TOTAL of MEMBERS.G, TOTAL of MEMBERS.P, TOTAL of
MEMBERS.T)
Rpr down PROD F.SALES_ZERO
PROD F.SALES_ZERO
PROD_BIKE 10
PROD_CAR 20
PROD_WAGON 0
MYCUSTAGG 30
```

Don't translate NA to ZERO, to show there is no data for PROD_WAGON in the SALES cube

```
dfn F.SALES_NOZERO formula dec <PROD GEOG TIME>
eq Eq MCALC(v.sales, SCREENBY if V.SALES eq NA THEN NO else YES, TOTAL of
MEMBERS.G, TOTAL of MEMBERS.P, TOTAL of MEMBERS.T)
rpr down PROD F.SALES_NOZERO
PROD F.SALES_NOZERO
PROD_BIKE 10
PROD_CAR 20
```

```
PROD_WAGON NA
MYCUSTAGG NA
```

Treat NA as ZERO only for custom aggregates by changing the formula so the MCALC function is not required until a custom aggregate is calculated

```
Dfn f.sales formula dec <PROD GEOG TIME>
eq if MEMBERS.P ne na or MEMBERS.G ne na
or MEMBERS.T ne na -
then F.SALES_ZERO else V.SALES
```

If a custom aggregate is not being calculated, the formula returns the data from the variable, otherwise MCALC calculates the custom aggregate treating NA as ZERO.

WORKSHEETS

Worksheets are an old Express feature originally used to import and export data to Lotus spreadsheets. While they have not been enhanced to handle Excel or other spreadsheets, they have some interesting and useful properties.

A worksheet is a two-dimensional object. Unlike other Express objects, each worksheet cell can contain different data types. The WKSDATA function is used to determine which data type is in each cell. Worksheets are useful to buffer or store data when you are uncertain of the data type.

TIME DIMENSIONS

Usually, time dimensions are just standard text dimensions that use application code to enforce special rules. When the administrator defines a time dimension, a series of objects are created that allow Express to properly calculate time series data (LAG, YTD). However, Express has the ability to define ‘true’ time dimensions that have an implicit knowledge of time. These time dimensions are broken into homogeneous pieces, with a dimension for each level in a hierarchy (such as MONTH, QUARTER and YEAR).

Time dimensions have many properties that an application designer might wish to use. Often, time dimensions are used to control and populate traditional text dimensions for reporting time periods. Time dimensions have implicit relations: a developer can limit months to year without a defined relation.

For example,

```
Limit YEAR to ‘YR99’
Limit MONTH to YEAR
```

will result in the 12 months of 1999.

Because true time dimensions are homogenous, they only store one level of data, and proper order is maintained you can easily find equivalent periods using positional logic. For example, to find a time period one year ago,

```
Limit month to ‘DEC99’
Limit month to month-12
```

Using this method, the code returns DEC98. Compare this to using a text time dimension, which is heterogeneous. Unless you are careful with your code, 12 members back could be any time period.

Sometimes time hierarchies are not easily derived from the source system, or time information such as Day of Week is not available. It can be faster to simply maintain the time dimensions in Express. For maintaining a text time dimension, Express has a number of useful functions with implicit relations describing time dimensions, including:

DAYOF: Day of Week for a date
DDOF: Day of Month for a date
MMOF: Month of a date

Using time dimensions, we can also design smaller and more efficient databases using TCONVERT.

TCONVERT

The TCONVERT function allows data to be aggregated or allocated from one time dimension to another, thereby reducing the size of the database considerably. For example, a sales system with two years of monthly level data normally requires 8 quarters and 2 years of pre-calculated data, making a time dimension of 34 members. Using TCONVERT, we reduce the size of the dimension to 24-month members, with a corresponding reduction in size of the cubes. In addition, TCONVERT supports average and last value as aggregation methods.

Although allocations are rarely used, TCONVERT also supports spreading data down to lower levels from higher levels. This can be useful if quota is set at the quarter level but users want to compare their current sales against their quarterly quotas.

Using TCONVERT adds some complexity to the database, but the complexity can be completely hidden from the end users. Continuing with above sales example, we can define the following:

```
Define MONTH dimension month
Define QUARTER dimension quarter
Define YEAR dimension year
Define V.SALES variable decimal <PRODGEOG <PROD GEOG> MONTH>
Ld Where the data is stored
Define Q.SALES formula decimal <PRODGEOG <PROD GEOG> QUARTER>
Eq TCONVERT(V.SALES, QUARTER, SUM)
```

```
Define Y.SALES formula decimal <PRODGEOG <PROD GEOG> YEAR>
Eq TCONVERT(V.SALES, YEAR, SUM)
```

None of these objects will be visible to the end user. Define the following formula for the end user:

```
Define F.SALES formula decimal <PROD GEOG TIME>
eq if isvalue(TIME YEAR) then Y.SALES(YEAR TIME ) else
If isvalue(TIME QUARTER) then Q.SALES(QUARTER TIME)
else if ISVALUE(TIME MONTH) then V.SALES(MONTH TIME)else na
```

Now, users have a text time dimension with embedded totals, but the aggregate numbers are calculated dynamically. Performance of TCONVERT is directly related to the amount of data being calculated. For example, a year value calculated from months is considerably faster than one calculated from days.

The formula that presents the data to the end user has little or no overhead. Both F.SALES and V.SALES demonstrate identical performance characteristics for stored data.

Capstones

Currently, Express is not a multi-processor database. Capstones are a technique of physically partitioning the databases to allow parallel processing. Although capstones add to the database complexity, there can be vast performance improvements.

There are two issues in using capstones as a design technique: how to physically separate the data and how to logically integrate it for transparent reporting.

Physically separating the data is the most challenging. However, there are some simple guidelines to follow.

- Split the data to avoid duplication of derived or input data.
- Split measures among databases only when the databases have different dimensionality or different densities.
- Split by time periods only when forced to by tight load windows.

Integrating the data requires a fairly simple set of formulas. The formulas are very similar to those given in the TCONVERT example above for creating a user accessible formula. Use the ISVALUE function on each dimension to determine the data’s location. If we assume the SALES data is split into two pieces, we can use the following formula:

```
Dfn F.SALES FORMULA DECIMAL <PROD GEOG TIME>
Eq if isvalue(DB1_PROD PROD) and isvalue(DB1_GEOG GEOG and isvalue(DB1_TIME
TIME)
then DB1_SALES(DB1_PROD PROD DB1_GEOG GEOG DB1_TIME TIME)
else if isvalue(DB2_PROD PROD) and isvalue(DB2_GEOG GEOG) and isvalue
(DB2_TIME TIME)
```

```
then DB2_SALES(DB2_PROD PROD DB2_GEOG GEOG DB2_TIME TIME)
else na
define DB1_SALES VARIABLE DECIMAL <DB1_PROD DB1_GEOG DB1_TIME>
define DB2_SALES VARIABLE DECIMAL <DB2_PROD DB2_GEOG DB2_TIME>
```

No matter how the databases are split, there appears to be no performance penalty using this approach.

In larger applications utilizing both TCONVERT and capstone functions, placing the TCONVERT formulas in the data databases and not in the capstone database will provide the best performance.

XCA

Express Communication Architecture (XCA) is Express' original method of communicating between a client and a server. With the release of Express Server 5.0, this communication method was replaced with Structured N-Dimensional Programming Interface (SNAPI). However, XCA has the following useful features:

- Using the pipeline option, you can move database objects directly from database to database without resorting to EIF files.
- Server to server connections can be established, allowing data, programs, and files to be moved from server to server.
- Using Personal Express, data can be sliced down to fit on a client machine and run in disconnect mode.

In Express Server 6.2 a new set of XCA functions were released to support Oracle Financial Analyzer.

EXECSTART allows you to issue a command to another Express database while continuing to process the first database. Using XCA you can attach another database and move a cube of data. EXECSTART will allow you to rollup the data and then bring the calculated data back to the primary database.

A code fragment using this technique follows:

```
cd /databasedir
dtb attach data.db rw
" Initiate the secondary session
comset type local "Connect to the same machine you are on
connect
xopen "Open XCA port
" In the secondary session, attach database read-only
execute 'cd /databasedir
execute 'dtb ro data.db'
" In the secondary session, rollup sales dollar variable
execstart 'call ROLLUP_PRG(\'SALES_DOL\')'
```

```
" At the same time rollup the sales unit variable
call rollup_prg('SALES_UNT')
upd
" In the primary session, attach and detach the
" database in order to clear Express memory
dtb detach acme
dtb attach acme rw
" Wait for the secondary rollup. Test for errors.
trap on errwait
xca_err = execwait
trap off
" In the secondary session, export the sales dollar cube back to the primary
session
execute 'export SALES_DOL to eif pipeline'
upd
dtb detach acme
xclose
disconnect
return
errwait:
"Handle the execwait error
return
END
```

The advantage of XCA over capstones is that the database design and code stream is simpler. However, moving the data from the secondary session to the primary session can be slow. This process works best with a larger number of smaller cubes.

You can create more than one secondary session by specifying a session number. You should only create as many sessions as you have available CPU’s and adequate memory to support the process.

Faster SQL-Based Data Loaders

Express can read data from a relational database. However, it can be slower than reading from flat files because Express requires two passes through the data. The first pass maintains the dimensions and the second pass loads the data. Skipping the two-pass approach will save some processing time but will cause the database to become disorganized.

If performance is an issue, you can write a SQL-based data load program that makes one pass through the data. The program performs all the dimension maintenance and buffers the data. As a post process in the program, the data is copied into the cubes, avoiding a second pass. My benchmarking indicates that this is as fast as creating a data file and writing a traditional flat file loader.

Below is a code example:

```
"|-------------------------------------------------------------------------"|
"| Name: READ_SALES
"|
"| Description: Simple example of buffered SQL data loader
"|
"| Author: WGBROWN (Symmetry)
"|
"| Date: 05/07/98
"|
"| Date Who Change
"| ------ ---- ------------------------------------------------------------
"|
"| ------------------------------------------------------------------------
arg _month date
vrb _time text
vrb _geog text
vrb _prod text
vrb _cnt int
vrb _sec int
vrb _sales dec
trap on error
call prg_init('READ_SALES')
pushlevel 'READ_SALES'
trap on error
"-------------------------------------------------------------------------
" Define all the buffer objects required for the database
if not exists('REC')
then define REC DIMENSION INT database BASE
if not exists('REC_MONTH')
then define REC_MONTH relation MONTH <REC> DATABASE BASE temp
if not exists('REC_STORE')
then define REC_STORE RELATION STORE <REC> DATABASE BASE temp
if not exists('REC_PROD')
then define REC_PROD RELATION STORE <REC> DATABASE BASE temp
if not exists('REC_SALES')
then define REC_SALES DEC <REC> DATABASE BASE temp
"add a 5000 dimension members to store the data.
mnt rec dlt all
mnt rec add 5000
"-------------------------------------------------------------------------
```

```
call prg_msgs(joinchars('Starting SALES for '_day))
_cnt=0
_sec=seconds
sql declare c1 cursor for -
select MONTH, GEOG, PROD, SALES -
from SALES where MONTH = :_month
order by MONTH, GEOG, PROD
if sqlcode ne 0
then signal sqlerr 'Cursor declare failed'
sql open c1
call prg_msgs(joinchars('Reading Sales, Cursor opened'))
while sqlcode eq 0
do
sql fetch c1 into :_time, :_geog, :_prod, :_sales
if _geog eq na
then goto skip
_cnt = _cnt+1 "record counter
if not isvalue(MONTH _time)
then signal 'BADTIME' 'Invalid time period in READ_SALES'
mnt GEOG merge _geog
mnt PROD merge _prod
REC_DAY (REC _CNT) = _time
REC_GEOG (REC _cnt) = _geog
REC_PROD (REC _CNT) = _prod
REC_SALES(REC _CNT) = _sales
if rem(_cnt 5000) eq 0
then do
update
call prg_msgs(joinchars('Read in '_cnt ' records in 'seconds-_sec))
mnt rec add 5000
doend
skip:
doend
call prg_msgs(joinchars('Read in '_cnt ' records in'seconds-_sec))
sql close c1
sql rollback work
"---------------------------------------------------------------
"Move data from buffers to the real cubes
limit.sortrel=n
oknullstatus=y
limit TIME to rec_month ifnone done
for TIME "For each time member that has been read into the buffedo
```

```
limit rec to TIME "Select the months records
call prg_msgs(joinchars('Starting assign of data for ' TIME)
"Sum the data from the buffer into the cube.
SALES = total(REC_SALES, REC_GEOG REC_PROD)
update
DATA_LOADED(MEASURE 'SALES')=YES "set data loaded flag for reference
call prg_msgs(joinchars('Finished assign of data for ' MONTH)
update
doend
done:
call prg_done
return
error:
sql close c1
sql rollback work
signal errorname errortext
return
```

Custom Functions

Functions are Express routines that take an argument and return a result. There are hundreds of functions built into Express and you can expand on that list by writing your own functions. Writing custom functions is quite easy in Express.

Formulas allow complex calculations that act on two or more cubes, or act along a dimension. For example, we can compute variance as ACTUALS-BUDGET and a 3 month moving average as movingaverage(f. sales, -3, 1, 1).

We can also combine the flexibility of formulas with the power of functions. We can write a function that performs a more complex calculation than a formula and can assign the function to the formula definition. For example:

```
Define F.SALES formula decimal <PROD GEOG TIME>
Eq f.sales.prg
Dfn F.SALES.PRG
Prg
Vrb result dec
"Simple program to calculate data on the fly
"Data is stored in a different cube DATA_SALES, with a relation
"linking PROD to D_PROD, GEOG to D_GEOG, etc.
limit D_PROD to PROD_DATA
limit D_GEOG to GEOG_DATA
limit D_TIME to TIME_DATA
```

```
result = total(DATA_SALES, -
DATA_PROD PROD DATA_GEOG GEOG DATA_TIME TIME)
return result
end
```

Assigning a newly created function to a formula definition is a very useful and powerful technique. It is critical to note that performance is directly related to the complexity and amount of data the function is calculating.

Conjoints

Conjoints were originally developed to control data sparsity. In the 6.x releases of Express, composites were developed to replace conjoints. While conjoints have slipped away from the developer's toolkit, they still have numerous uses.

Many-to-Many Relations

OLAP applications often require establishing a relationship between two dimensions. Perhaps the most common is the relationship between a company and its currency for performing exchange rate calculations. The object to support this might be defined as follows:

```
define COMPANY_CURRENCY relation CURRENCY <COMPANY>
```

This will work as long as each company has one and only one currency (which is usually the case).

However, if the need for a many-to-many relationship arises, you can create one using a conjoint. For example, each department could have employees with many titles, and the same titles could be used throughout the company. While this is impossible to create with the standard relation object, with a conjoint, a many-to-many relationship can be defined like this:

```
define DEPT_TITLE dimension <DEPT TITLE>
```

If you also need to know that titles are valid in a department, the commands are:

```
limit DEPT to 'XXX'
limit DEPT_TITLE to DEPT
limit TITLE to DEPT_TITLE
```

The status of the title dimension contains the valid members for department XXX.

Performance

While composites are the simplest method of controlling data sparsity, conjoints offer greater control and, in some cases, better performance. While loading data into a conjoint is fairly well documented, Oracle

does not document the method to aggregate data along a conjoint. After loading the input level data, the embedded totals (also known as aggregate levels) and relations for the ROLLUP command still need to be created and maintained.

Here is a code fragment that will maintain one dimension of a conjoint. To maintain each subsequent dimension, just repeat the code.

```
"Define the relationship to rollup the geography dimension over the list of
geography "hierarchies
"GEOGPROD is the conjoint, GH is the list of geography hierarchies.
if not exists('GEOG_GHIER')
then define GEOG_GHIER relation GEOGPROD <GEOGPROD GH>
for GH "For each GEOG hierarchy
do
limit GEOG to all "Select all geographies
while statlen(GEOG) ne 0
do
"Each loop will step up the hierarchy until reaching the top (statlen is
equal to 0)
limit GEOG remove ancestors using PRNTREL.G
limit GEOGPROD to GEOG "Only loop over the required geographies
"Add parent level (stored in PRNTREL.G) while filling in relationship
mnt GEOGPROD merge <PRNTREL.G(GEOG key(GEOGPROD GEOG)) KEY(GEOGPROD PROD)>
relate GEOG_GHIER
"Set status to next level, if statlen equal 0 then stop, else continue
limit GEOG to PRNTREL.G
doend "statlen ne 0 loop
doend "Loop over GH
```

You can aggregate data over the conjoint by using the conjoint parent-child relationship illustrated below (use one ROLLUP command for each dimension in the conjoint).

```
rollup V.SALES over GEOGPROD using GEOG_GHIER
```

If your application requires the additional functionality of composites, but you want to speed up the aggregation process using conjoints, you can switch back and forth between them quite simply.

Use the CHGDFN command as follows to change from composite to conjoint, and back again:

```
chgdfn GEOGPROD composite "Turns GEOGPROD to a composite
chgdfn GEOGPROD dimension "Returns GEOGPROD to a dimension
```

One important difference between composites and conjoints is that composites can only be used with variables. In the example below, any other object dimensioned by GEOGPROD must first be deleted before issuing the CHGDFN command. For example, after running the conjoint maintenance code above, the following commands must be issued before changing the conjoint back to a composite:

```
if exists('GEOG_GHIER')
then delete GEOG_GHIER database &obj(database 'GEOG')
chgdfn GEOGPROD composite
```

Leveraging Express

As you can see, Express' rich stored procedure language can help you develop creative solutions to a wide range of business problems. By following the tips and techniques summarized in this article, you will be on your way to taking advantage of Express' power and flexibility.